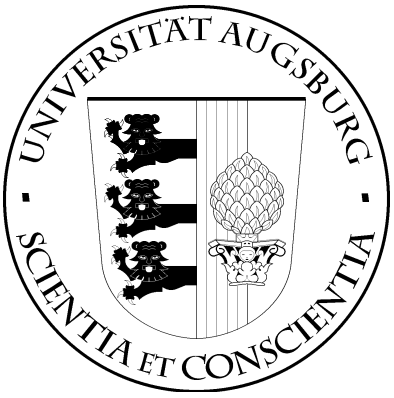
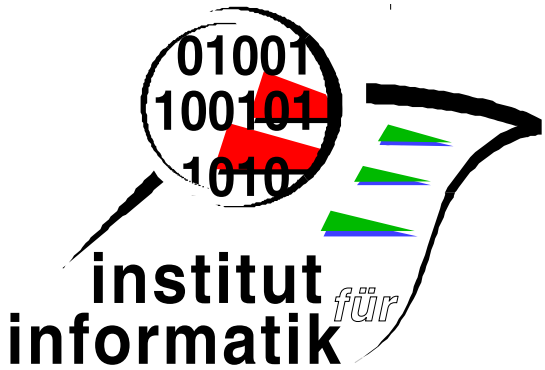
Universität Augsburg



The Confidence-Probability Semiring implemented within `OpenFst`

Markus Huber and Christian Kölbl

Report 2011-16 March 2012



Institut für Informatik
D-86135 Augsburg

Introduction

In [6] the authors were involved in the construction of the so-called *confidence-probability semiring* $(\mathbb{K}^\circ, \oplus, (-\infty, \infty), \otimes, (0, 0))$. Every element $(c, p) \in \mathbb{K}^\circ$ is a tupel consisting of a *confidence* c, originating from the arctic semiring, and a *probability* p, originating from the tropical semiring. As described in [2], [3] and [4] these values can be used in modelling uncertain information within dialogue systems.

In this report the implementation of the semiring is described. It builds on top of the `OpenFst`-library [1] and as a direct implication the language used is C++. The library implements data structures and algorithms for *finite state transducers* [5].

Within the `OpenFst`-library the tropical semiring is already implemented, whereas the arctic semiring is missing. Therefore the arctic semiring is implemented first.

The basic set of the confidence-probability semiring does not include tupels with negative numbers. Because of this issue special classes representing confidence weight and probability weight are constructed.

After this groundwork the final confidence-probability weight can be implemented.

Arctic Semiring

On the basis of the implementation of the tropical semiring taken from the file `float-weight.h` of the `OpenFst` distribution the arctic semiring is implemented as a template class inheriting from `FloatWeightTpl<T>`. First the layout of the class is given.

1 ⟨*Layout of class* `ArcticWeightTpl<T>` 1⟩≡ (7b)

```
template<class T>
class ArcticWeightTpl : public FloatWeightTpl<T>
{
public:
  typedef ArcticWeightTpl<T> WeightType;
  typedef FloatWeightTpl<T> BaseType;

  ⟨Constructors of class ArcticWeightTpl<T> 3b⟩
  ⟨Member function defining the type of the semiring (arctic weight) 2a⟩
  ⟨Stuff specific to OpenFst (arctic weight) 6c⟩
  ⟨Properties of the arctic semiring 6a⟩
  ⟨Member functions defining the basic set (arctic weight) 2b⟩
```

⟨*Member functions giving the identities* *(arctic weight)* 3a⟩
};

To save some typing two `typedef`s are introduced which are used throughout the remaining implementation.

All tools from `OpenFst` use a string to identify which semiring to use. The arctic semiring should be no exception. The string is composed in analogy to the tropical semiring and the method is implemented the same way using a `static` variable and returning a reference to this variable.

2a ⟨*Member function defining the type of the semiring* *(arctic weight)* 2a⟩≡ (1)

```
static const std::string &Type()
{
  static const std::string type =
    "arctic" + BaseType::GetPrecisionString();

  return type;
}
```

The basic set for the arctic semiring is $\mathbb{R} \cup \{-\infty\}$; following the basic set $\mathbb{R} \cup \{\infty\}$ of the already implemented tropical semiring. Now a method `Member()` has to be constructed returning `true` iff the return value of `Value()` – from the base class – lies within the basic set.

2b ⟨*Member functions defining the basic set* *(arctic weight)* 2b⟩≡ (1) 2c▷

```
bool Member() const
{
  return Value() == Value() &&
         Value() != FloatLimits<T>::kPosInfinity;
}
```

The expression evaluates to true if the contained number is a real number or the value $-\infty$. If the number is the result of a division by zero it has the value *NaN* and the first comparison evaluates to `false`.

An additional function defines an entity which is not a member of the basic set. The implementation is taken from the one of the tropical semiring.

2c ⟨*Member functions defining the basic set* *(arctic weight)* 2b⟩+≡ (1) ◁2b

```
static const WeightType NoWeight()
{
  static const WeightType no(FloatLimits<T>::kNumberBad);

  return no;
}
```

After the basic set we implement the identity elements. For every identity a separate `static` method has to be defined. Each method returns the respective element.

The multiplicative identity of the semiring is the element 0 whereas the additive identity is the element $-\infty$. In order to avoid constructing the objects every time the methods are called, they are defined as `static` variables within the methods and in each case a reference to the variable is returned to prevent calling the copy constructors. The implementation closely resembles the one taken from the file `float-weight.h` of the `OpenFst` distribution.

3a ⟨*Member functions giving the identities (arctic weight)* 3a⟩≡ (1)

```
static const WeightType &One()
{
  static const WeightType one(0.0F);

  return one;
}

static const WeightType &Zero()
{
  static const WeightType zero(FloatLimits<T>::kNegInfinity);

  return zero;
}
```

Now the constructors for the class are defined. A constructor taking a single floating number was already used and is therefore needed. Additional a default constructor and a copy constructor are defined. All constructors call the appropriate constructors from the base class `FloatWeightTpl<T>`. Besides the calling they do nothing else.

3b ⟨*Constructors of class* `ArcticWeightTpl<T>` 3b⟩≡ (1)

```
ArcticWeightTpl() :
  BaseType()
{
}

ArcticWeightTpl(T f) :
  BaseType(f)
{
}
```

```
ArcticWeightTpl(const WeightType &w) :
  BaseType(w)
{
}
```

As a next step the operations of the semiring are defined. `OpenFst` requires the implementation of addition, multiplication, and division if possible. They all are implemented outside the class.

The arctic addition is simply the maximization. Because $-\infty < \mathtt{w}$ holds for all floating point numbers `w` in C++ no case-by-case analysis is needed.

4a ⟨*Addition (arctic weight)* 4a⟩≡ (7b)

```
template<class T>
inline ArcticWeightTpl<T> Plus(const ArcticWeightTpl<T> &w1,
                               const ArcticWeightTpl<T> &w2)
{
  return w1.Value() < w2.Value() ? w2 : w1;
}
```

The arctic multiplication is even simpler: it is the well known addition. But this time cases are needed. First references to the floating point numbers are defined to save some typing.

4b ⟨*Multiplication (arctic weight)* 4b⟩≡ (7b) 4c▷

```
template<class T>
inline ArcticWeightTpl<T> Times(const ArcticWeightTpl<T> &w1,
                                const ArcticWeightTpl<T> &w2)
{
  const T &f1 = w1.Value(), &f2 = w2.Value();
```

Now we have to check if one of the factors is the identity and if so return the other factor immediately. We do not test on the abstraction layer of the arctic semiring but on the layer of floating point numbers. This way no additional function calls and no objects are involved.

4c ⟨*Multiplication (arctic weight)* 4b⟩+≡ (7b) ◁4b 5a▷

```
  if (f1 == FloatLimits<T>::kNegInfinity) return w2;
  if (f2 == FloatLimits<T>::kNegInfinity) return w1;
```

If neither the first nor the second factor was the identity the simple sum can be returned. The return value is an object that has to be constructed. If we constructed the object inside the method this object would have been copied during the `return` resulting in constructing a second object. So we

simply return the sum of the two floating point numbers and only one object is constructed.

5a ⟨*Multiplication (arctic weight)* 4b⟩+≡ (7b) ◁4c

```
  return f1 + f2;
}
```

Now let us take a look at the division operation. The file `weight.h` states that for all `a` and `c` for which the method `Member()` evaluates to `true`

$$\texttt{Divide(c, a)} = \texttt{b} \wedge \texttt{Times(a, b)} = \texttt{c} \wedge \texttt{Times(b, a)} = \texttt{c}$$

where `Member(b)` also evaluates to `true` should hold, if the semiring is commutative. Additionally there should be no difference, if `Divide()` is called with a third parameter or not.

As the arctic semiring is commutative the function `Divide()` has to be implemented to fullfill the requests. The function `Times()` already shows a commutative behaviour as the addition of floating point numbers is commutative. So division is basically subtraction but we take special care of two cases. First division by zero returns a value that is not a number and second if zero should be divided zero is returned. As with the function `Times()` we test on a lower level than the semiring abstraction.

5b ⟨*Division (arctic weight)* 5b⟩≡ (7b)

```
template<class T>
inline ArcticWeightTpl<T> Divide(const ArcticWeightTpl<T> &w1,
                                 const ArcticWeightTpl<T> &w2,
                                 DivideType typ = DIVIDE_ANY)
{
  const T &f1 = w1.Value(), &f2 = w2.Value();

  if (f2 == FloatLimits<T>::kNegInfinity)
    return FloatLimits<T>::kNumberBad;
  if (f1 == FloatLimits<T>::kNegInfinity)
    return FloatLimits<T>::kNegInfinity;

  return f1 - f2;
}
```

After defining all of the operations we can eventually set up the method giving the properties of the arctic semiring. Again the file `weight.h` states

the meaning of the different properties.

6a ⟨*Properties of the arctic semiring* 6a⟩≡ (1)

```
static uint64 Properties()
{
  return kLeftSemiring | kRightSemiring | kCommutative |
         kIdempotent | kPath;
}
```

Again in analogy to the tropical semiring we give a `typedef` to specify the standard weight type and add hardcoded versions of the operations for the types `float` and `double` which actually just call the templated versions.

6b ⟨*Standard type and operations (arctic weight)* 6b⟩≡ (7b) 23a▷

```
// Single precision ArcticWeight
typedef ArcticWeightTpl<float> ArcticWeight;

inline ArcticWeightTpl<float> Plus(const ArcticWeightTpl<float> &w1,
                                   const ArcticWeightTpl<float> &w2)
{
  return Plus<float>(w1, w2);
}

inline ArcticWeightTpl<float> Times(const ArcticWeightTpl<float> &w1,
                                    const ArcticWeightTpl<float> &w2)
{
  return Times<float>(w1, w2);
}

inline ArcticWeightTpl<float> Divide(const ArcticWeightTpl<float> &w1,
                                     const ArcticWeightTpl<float> &w2,
                                     DivideType type = DIVIDE_ANY)
{
  return Divide<float>(w1, w2);
}
```

For type `double` the same three functions are needed but they are given in the appendix.

As for the tropical semiring we need some more stuff specific to `OpenFst` to provide full functionality. These things are copied directly from the file `float-weight.h`.

6c ⟨*Stuff specific to* `OpenFst` *(arctic weight)* 6c⟩≡ (1) 7a▷

```
using BaseType::Value;

typedef WeightType ReverseWeight;

ReverseWeight Reverse() const
{
  return *this;
}
```

7a ⟨*Stuff specific to* `OpenFst` *(arctic weight)* 6c⟩+≡ (1) ◁6c

```
WeightType Quantize(float delta = kDelta) const
{
  if (Value() == FloatLimits<T>::kNegInfinity ||
      Value() == FloatLimits<T>::kPosInfinity ||
      Value() != Value())
    return *this;
  else
    return floor(Value()/delta + 0.5F) * delta;
}
```

Now we are able to put it all together and give the whole file `arctic-weight.h`.

7b ⟨*Programme/arctic-semiring/arctic-weight.h* 7b⟩≡

```
#ifndef _ARCTIC_WEIGHT_H_INCLUDED
#define _ARCTIC_WEIGHT_H_INCLUDED

#include <fst/float-weight.h>

namespace fst
{
  ⟨Layout of class ArcticWeightTpl<T> 1⟩

  ⟨Addition (arctic weight) 4a⟩
  ⟨Multiplication (arctic weight) 4b⟩
  ⟨Division (arctic weight) 5b⟩

  ⟨Standard type and operations (arctic weight) 6b⟩
}

#endif
```

The semiring should be compiled into a dynamic library which can then be loaded by the tools from `OpenFst` to work with weights from the arctic semiring.

Programmes written by oneself should not need to load the library but include a special header file instead. This header mainly contains another `typedef` which combines a templated class for arcs with the new weights. This is needed because the FST-types from `OpenFst` are used with types of arcs and not types of weights.

This header file is given now.

8a ⟨*Programme/arctic-semiring/arctic-arc.h* 8a⟩≡

```
#ifndef _ARCTIC_ARC_H_INCLUDED
#define _ARCTIC_ARC_H_INCLUDED

#include <fst/arc.h>
#include "arctic-weight.h"

namespace fst
{
  typedef ArcTpl<ArcticWeight> ArcticArc;
}

#endif
```

For the dynamic library another file is needed which basically registers the new type of arcs for use with the different types of FSTs. The exact syntax is taken from the forum found on the homepage of the `OpenFst` project.

8b ⟨*Programme/arctic-semiring/arctic.c++* 8b⟩≡

```
#include <fst/fst.h>
#include <fst/const-fst.h>
#include <fst/script/register.h>
#include <fst/script/fstscript.h>

#include "arctic-arc.h"

namespace fst
{
  namespace script
  {
    REGISTER_FST(VectorFst, ArcticArc);
    REGISTER_FST(ConstFst, ArcticArc);
```

```
    REGISTER_FST_CLASSES(ArcticArc);
    REGISTER_FST_OPERATIONS(ArcticArc);
  }
}
```

The library's name should be `arctic-arc.so` on Unix systems and `arctic-arc.dll` on Win32 systems.

Confidence-Probability Semiring

Before we start giving the implementation of the confidence-probability semiring we first construct two special classes for the confidence respectively probability part of the tupels in question.

Confidence Weight

As stated in the introduction the confidence part comes from the arctic semiring. There is only one little difference. A confidence cannot be equal to or less than zero. Consequently the method `Member()` for the class `ConfidenceWeightTpl<T>` looks like the following.

9a ⟨*Member functions defining the basic set (confidence weight)* 9a⟩≡ (11c) 9b▷

```
bool Member() const
{
  return BaseType::Member() &&
         Value() > 0;
}
```

However there is one exception: A confidence can have the value $-\infty$ iff it is part of the additive identity of the confidence-probability semiring. This exception will be covered in the class `ConfidenceProbabilityWeightTpl<T>`.

Again we need to define a method which returns en entity that is not an element of the basic set. We use the method from the base class.

9b ⟨*Member functions defining the basic set (confidence weight)* 9a⟩+≡ (11c) ◁9a

```
static const WeightType &NoWeight()
{
  static const WeightType no(BaseType::NoWeight());

  return no;
}
```

The identity elements remain the same. However to get methods that return the right type we have to redefine them using the methods from the base class. Again we use the `typedef WeightType` to save space and typing within the class.

10a ⟨*Member functions giving the identities (confidence weight)* 10a⟩≡ (11c)

```
static const WeightType &One()
{
  static const WeightType one(BaseType::One());

  return one;
}

static const WeightType &Zero()
{
  static const WeightType zero(BaseType::Zero());

  return zero;
}
```

To be able to construct an object of type `ConfidenceWeightTpl<T>` from an object of type `ArcticWeightTpl<T>` as used by those three methods we have to define an additional constructor. We declare it `explicit` so it does not interfere with the type system.

The other three constructors follow the pattern already used by class `ArcticWeightTpl<T>`.

10b ⟨*Constructors of class* `ConfidenceWeightTpl<T>` 10b⟩≡ (11c)

```
ConfidenceWeightTpl() :
  BaseType()
{
}

ConfidenceWeightTpl(T f) :
  BaseType(f)
{
}

ConfidenceWeightTpl(const WeightType &w) :
  BaseType(w)
{
}
```

```
explicit ConfidenceWeightTpl(const BaseType &w) :
  BaseType(w)
{
}
```

For defining the operations of the confidence-probability semiring we need two more methods within the class `ConfidenceWeightTpl<T>`. These two operations are a comparison and an addition.

11a ⟨*Additional operations (confidence weight)* 11a⟩≡ (11c) 11b▷

```
inline bool operator<(const WeightType &w) const
{
  return Value() < w.Value();
}
```

This method will be used in implementing (9) from [6]. The equality also used in (9) is already defined in `float-weight.h`. However (9) is implemented after the class `ConfidenceProbabilityWeightTpl<T>` is defined.

The addition takes care of the special cases where one of the summands has the value $-\infty$. The method will be used in implementing (13) from [6].

11b ⟨*Additional operations (confidence weight)* 11a⟩+≡ (11c) ◁11a

```
inline WeightType operator+(const WeightType &w) const
{
  if (Value() == FloatLimits<T>::kNegInfinity ||
      w.Value() == FloatLimits<T>::kNegInfinity)
    return FloatLimits<T>::kNegInfinity;

  return Value() + w.Value();
}
```

Now the layout of the class `ConfidenceWeightTpl<T>` can be given.

11c ⟨*Layout of class* `ConfidenceWeightTpl<T>` 11c⟩≡ (12c)

```
template<class T>
class ConfidenceWeightTpl : public ArcticWeightTpl<T>
{
public:
  typedef ConfidenceWeightTpl<T> WeightType;
  typedef ArcticWeightTpl<T> BaseType;

  ⟨Constructors of class ConfidenceWeightTpl<T> 10b⟩
  ⟨Stuff specific to OpenFst (confidence weight) 12a⟩
```

```
  ⟨Member functions defining the basic set (confidence weight) 9a⟩
  ⟨Member functions giving the identities (confidence weight) 10a⟩
  ⟨Additional operations (confidence weight) 11a⟩
};
```

Again we need some stuff specific to `OpenFst` to provide full functionality and to be able to use the class `PairWeight<W1, W2>` in the implementation of class `ConfidenceProbabilityWeightTpl<T>`.

12a ⟨*Stuff specific to* `OpenFst` *(confidence weight)* 12a⟩≡ (11c) 12b▷

```
using BaseType::Value;

typedef WeightType ReverseWeight;

ReverseWeight Reverse() const
{
  return *this;
}
```

The method `Quantize()` uses the same method from the base class but cannot simply return its value because the according constructor was declared `explicit`. So a temporary object has to be constructed.

12b ⟨*Stuff specific to* `OpenFst` *(confidence weight)* 12a⟩+≡ (11c) ◁12a

```
WeightType Quantize(float delta = kDelta) const
{
  return WeightType(BaseType::Quantize(delta));
}
```

The last thing is to give the whole file `confidence-weight.h`.

12c ⟨*Programme/confidence-probability-semiring/confidence-weight.h* 12c⟩≡

```
#ifndef _CONFIDENCE_WEIGHT_H_INCLUDED
#define _CONFIDENCE_WEIGHT_H_INCLUDED

#include "arctic-semiring/arctic-weight.h"

namespace fst
{
  ⟨Layout of class ConfidenceWeightTpl<T> 11c⟩
}

#endif
```

Probability Weight

Now we give another class which implements the probability part. As mentioned in the introduction it comes from the tropical semiring. Its layout follows the one from the class `ConfidenceWeightTpl<T>`.

13a ⟨*Layout of class* `ProbabilityWeightTpl<T>` 13a⟩≡ (25a)

```
template<class T>
class ProbabilityWeightTpl : public TropicalWeightTpl<T>
{
public:
  typedef ProbabilityWeightTpl<T> WeightType;
  typedef TropicalWeightTpl<T> BaseType;

  ⟨Constructors of class ProbabilityWeightTpl<T> 23b⟩
  ⟨Stuff specific to OpenFst (probability weight) 24a⟩
  ⟨Member functions defining the basic set (probability weight) 14a⟩
  ⟨Member functions giving the identities (probability weight) 24c⟩
  ⟨Additional operations (probability weight) 13b⟩
};
```

The additional operations are again a comparison and an addition.

13b ⟨*Additional operations* (probability weight) 13b⟩≡ (13a) 13c▷

```
inline bool operator>=(const WeightType &w) const
{
  return Value() >= w.Value();
}
```

This method will also be used in implementing (9) from [6].

The addition takes care of the special cases where one of the summands has the value ∞. The method will be used in implementing (13) from [6].

13c ⟨*Additional operations* (probability weight) 13b⟩+≡ (13a) ◁13b

```
inline WeightType operator+(const WeightType &w)
{
  if (Value() == FloatLimits<T>::kPosInfinity ||
      w.Value() == FloatLimits<T>::kPosInfinity)
    return FloatLimits<T>::kPosInfinity;

  return Value() + w.Value();
}
```

Again there is a little difference between an element of the tropical semiring and a probability. As stated in (15) in [6], a probability cannot be less than zero. Consequently the method `Member()` looks like the following.

14a ⟨*Member functions defining the basic set (probability weight)* 14a⟩≡ (13a) 24b▷

```
bool Member() const
{
  return BaseType::Member() &&
    Value() >= 0;
}
```

The rest of the implementation is given in the appendix because it only resembles the one from class `ConfidenceWeightTpl<T>`.

Confidence-Probability Weight

Now we are ready to define the class `ConfidenceProbabilityWeightTpl<T>`. As mentioned before we utilize the template class `PairWeight<W1, W2>` from the `OpenFst` distribution. This class represents a tuple of weights and already defines some usefull code. So the layout of the new class looks like the following.

14b ⟨*Layout of class* `ConfidenceProbabilityWeightTpl<T>` 14b⟩≡ (19a)

```
template<class T>
class ConfidenceProbabilityWeightTpl :
  public PairWeight<ConfidenceWeightTpl<T>, ProbabilityWeightTpl<T> >
{
public:
  typedef ConfidenceProbabilityWeightTpl<T> WeightType;
  typedef ConfidenceWeightTpl<T> ConfidenceType;
  typedef ProbabilityWeightTpl<T> ProbabilityType;
  typedef PairWeight<ConfidenceType, ProbabilityType> BaseType;

  ⟨Constructors of class ConfidenceProbabilityWeightTpl<T> 15c⟩
  ⟨Member function defining the type of the semiring (confidence-probability weight) 16c⟩
  ⟨Stuff specific to OpenFst (confidence-probability weight) 15a⟩
  ⟨Properties of the confidence-probability semiring 16d⟩
  ⟨Member function defining the basic set (confidence-probability weight) 16b⟩
  ⟨Member functions giving the identities (confidence-probability weight) 16a⟩
  ⟨Additional operations (confidence-probability weight) 16e⟩
};
```

Again we use some `typedef`s to save space and typing.

Now we take a look at the member functions from the base class that we want to use. The methods `Value1()` and `Value2()` simply provide access to the values of the tupel. So we directly use them.

15a ⟨*Stuff specific to* ***OpenFst*** *(confidence-probability weight)* 15a⟩≡ (14b) 15b▷

```
using BaseType::Value1;
using BaseType::Value2;
```

The class `PairWeight<>` already defines the methods `Reverse()` and `Quantize()` the way we would. Again we simply use them.

15b ⟨*Stuff specific to* ***OpenFst*** *(confidence-probability weight)* 15a⟩+≡ (14b) ◁15a

```
typedef WeightType ReverseWeight;

using BaseType::Reverse;
using BaseType::Quantize;
```

By defining the `ReverseWeight`-type differently from the base class we need a special constructor which can convert an object from the base class to an object of the type dealt with herein. We immediately give the complete list of constructors. The last one is for the mentioned conversion.

15c ⟨*Constructors of class* `ConfidenceProbabilityWeightTpl<T>` 15c⟩≡ (14b)

```
ConfidenceProbabilityWeightTpl() :
  BaseType(Zero())
{
}

ConfidenceProbabilityWeightTpl(const WeightType &w) :
  BaseType(w)
{
}

ConfidenceProbabilityWeightTpl(const ConfidenceType &x1,
                               const ProbabilityType &x2) :
  BaseType(x1, x2)
{
}

ConfidenceProbabilityWeightTpl(const BaseType &w) :
  BaseType(w)
{
}
```

The default-constructor makes use of the method `Zero()`. Again we simply use the methods from the base class.

16a ⟨*Member functions giving the identities (confidence-probability weight)* 16a⟩≡ (14b)
```
using BaseType::Zero;
using BaseType::One;
```

The method `Member()` from the base class needs a little adjustment because the just defined identity elements contain entities which are not elements of the basic sets of the confidence weight respectively the probability weight anymore.

16b ⟨*Member function defining the basic set (confidence-probability weight)* 16b⟩≡ (14b)
```
bool Member() const
{
  return BaseType::Member() || *this == Zero() || *this == One();
}
```

The used `operator==()` is already defined by the base class.

As a last step we need to define the type and the properties of the confidence-probability semiring.

16c ⟨*Member function defining the type of the semiring (confidence-probability weight)* 16c⟩≡ (14b)
```
static const std::string &Type()
{
  static const std::string type = "confidence-probability";
  return type;
}
```

16d ⟨*Properties of the confidence-probability semiring* 16d⟩≡ (14b)
```
static uint64 Properties()
{
  return kLeftSemiring | kRightSemiring | kCommutative |
         kIdempotent;
}
```

Now we are ready to give the operations of the semiring. These operations will only contain addition and multiplication but no division because multiplication is not invertible in this semiring.

But before the addition can be formulated we need an implementation of `operator<=()` for the semiring. This directly implements (9) from [6].

16e ⟨*Additional operations (confidence-probability weight)* 16e⟩≡ (14b)
```
inline bool operator<=(const WeightType &w) const
{
```

```
    if (Value1() < w.Value1()) return true;
    if (Value1() == w.Value1() && Value2() >= w.Value2()) return true;

    return false;
  }
```

Now we can implement (10) from [6].

17a ⟨*Addition (confidence-probability weight)* 17a⟩≡ (19a) 17b▷

```
  template<class T>
  inline ConfidenceProbabilityWeightTpl<T>
    Plus(const ConfidenceProbabilityWeightTpl<T> &w,
         const ConfidenceProbabilityWeightTpl<T> &v)
  {
    if (w == ConfidenceProbabilityWeightTpl<T>::Zero()) return v;
    if (v == ConfidenceProbabilityWeightTpl<T>::Zero()) return w;
```

These two lines just make sure that the identity element is respected by the method `Plus()`.

17b ⟨*Addition (confidence-probability weight)* 17a⟩+≡ (19a) ◁17a

```
    if (v <= w) return w;
    if (w <= v) return v;

    LOG(FATAL) << "ConfidenceProbabilityWeight::Plus: "
                  "neither v <= w nor w <= v";

    return ConfidenceProbabilityWeightTpl<T>::Zero(); // should not happen!
  }
```

Now we have to define three more functions before we can give the multiplication function. First we implement the semiring-isomorphism μ from [6].

17c ⟨*Multiplication (confidence-probability weight)* 17c⟩≡ (19a) 18a▷

```
  template<class T>
  inline T mue(const ConfidenceWeightTpl<T> &c)
  {
    return 1 - std::exp(-c.Value());
  }
```

Second we need to give an `oprator*()` for an object of type `ConfidenceWeightTpl<T>` and an object of type `ProbabilityWeightTpl<T>`. This implements (4) from

[6] by exploiting the fact that the third and fourth case do not occur.

18a ⟨*Multiplication (confidence-probability weight)* 17c⟩+≡ (19a) ◁17c 18b▷

```
template<class T>
inline ProbabilityWeightTpl<T> operator*(const ConfidenceWeightTpl<T> &a,
                                         const ProbabilityWeightTpl<T> &x)
{
  if (a == ConfidenceWeightTpl<T>::Zero())
    return ProbabilityWeightTpl<T>::Zero();
  if (a == ConfidenceWeightTpl<T>::One())
    return ProbabilityWeightTpl<T>::One();

  return mue(a) * x.Value();
}
```

Third we have to give an `operator/` for an object of type `ProbabilityWeightTpl<T>` and an object of type `T`. This simply allows the fraction line in (13) from [6] to be written as `/`.

18b ⟨*Multiplication (confidence-probability weight)* 17c⟩+≡ (19a) ◁18a 18c▷

```
template<class T>
inline ProbabilityWeightTpl<T> operator/(const ProbabilityWeightTpl<T> &p,
                                         const T &c)
{
  return p.Value() / c;
}
```

Eventually we are ready to give the code for the multiplication.

18c ⟨*Multiplication (confidence-probability weight)* 17c⟩+≡ (19a) ◁18b 18d▷

```
template<class T>
inline ConfidenceProbabilityWeightTpl<T>
  Times(const ConfidenceProbabilityWeightTpl<T> &w,
        const ConfidenceProbabilityWeightTpl<T> &v)
{
  if (w == ConfidenceProbabilityWeightTpl<T>::One()) return v;
  if (v == ConfidenceProbabilityWeightTpl<T>::One()) return w;
```

Again these two lines asure that `Times()` respects the identity element.

18d ⟨*Multiplication (confidence-probability weight)* 17c⟩+≡ (19a) ◁18c

```
  ConfidenceWeightTpl<T> confidence(w.Value1() + v.Value1());
  ProbabilityWeightTpl<T> probability(w.Value1()*w.Value2() +
                                      v.Value1()*v.Value2());
```

```
  return ConfidenceProbabilityWeightTpl<T>(confidence,
                                           probability / mue(confidence));
}
```

As already explained during the implementation of the arctic semiring three files are needed. The first file contains all the definitions of the semiring class.

19a ⟨*Programme/confidence-probability-semiring/confidence-probability-weight.h* 19a⟩≡

```
#ifndef _CONFIDENCE_PROBABILITY_WEIGHT_H_INCLUDED
#define _CONFIDENCE_PROBABILITY_WEIGHT_H_INCLUDED

#include <fst/pair-weight.h>
#include "confidence-probability-semiring/confidence-weight.h"
#include "confidence-probability-semiring/probability-weight.h"
#include <cmath>

namespace fst
{
  ⟨Layout of class ConfidenceProbabilityWeightTpl<T> 14b⟩

  ⟨Addition (confidence-probability weight) 17a⟩
  ⟨Multiplication (confidence-probability weight) 17c⟩
  ⟨Division (confidence-probability weight) 25b⟩

  ⟨Standard type and operations (confidence-probability weight) 19b⟩
}

#endif
```

The standard type is called `SemanticWeight`.

19b ⟨*Standard type and operations* (confidence-probability weight) 19b⟩≡ (19a) 26▷

```
typedef ConfidenceProbabilityWeightTpl<float> SemanticWeight;
```

The operations are given in the appendix. Again they are simple instatiations of the template functions for the types `float` and `double`.

The header file for use in new applications is given here.

19c ⟨*Programme/confidence-probability-semiring/confidence-probability-arc.h* 19c⟩≡

```
#ifndef _CONFIDENCE_PROBABILITY_ARC_H_INCLUDED
#define _CONFIDENCE_PROBABILITY_ARC_H_INCLUDED
```

```
#include <fst/arc.h>
#include "confidence-probability-semiring/confidence-probability-weight.h"

namespace fst
{
  typedef ArcTpl<SemanticWeight> SemanticArc;
}

#endif
```

And in order to be able to compile the second shared library we need to have another file.

20a ⟨*Programme/confidence-probability-semiring/confidence-probability.c++* 20a⟩≡

```
#include <fst/fst.h>
#include <fst/const-fst.h>
#include <fst/script/register.h>
#include <fst/script/fstscript.h>

#include "confidence-probability-semiring/confidence-probability-arc.h"

namespace fst
{
  namespace script
  {
    REGISTER_FST(VectorFst, SemanticArc);
    REGISTER_FST(ConstFst, SemanticArc);
    REGISTER_FST_CLASSES(SemanticArc);
    REGISTER_FST_OPERATIONS(SemanticArc);
  }
}
```

Now it is possible to compile the confidence-probability semiring and use it with `OpenFst` .

Makefile

To ease the process of compilation we provide this `Makefile`.

20b ⟨*Programme/Makefile* 20b⟩≡

```
aDIR  = arctic-semiring
cbDIR = confidence-probability-semiring
```

```
aSRC  = $(aDIR)/arctic-weight.h
aSRC += $(aDIR)/arctic-arc.h
cbSRC = $(cbDIR)/confidence-probability-weight.h
cbSRC+= $(cbDIR)/confidence-probability-arc.h
cbSRC+= $(cbDIR)/confidence-weight.h
cbSRC+= $(cbDIR)/probability-weight.h

ALL: arctic-arc.so confidence_probability-arc.so

.PHONY: ALL

arctic-arc.so: $(aDIR)/arctic.c++ $(aSRC)
  g++ -Os -shared -fPIC $< -o $@ -I./
  strip --strip-unneeded $@

confidence_probability-arc.so: $(cbDIR)/confidence-probability.c++ $(cbSRC)
  g++ -Os -shared -fPIC $< -o $@ -I./
  strip --strip-unneeded $@
```

References

[1] Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri. Openfst: A general and efficient weighted finite-state transducer library. In *Proceedings of the Twelth International Conference on Implementation and Application of Automata, (CIAA 2007)*, volume 4783 of *Lecture Notes in Computer Science*, pages 11–23. Springer, 2007.

[2] Markus Huber. *Semantische Informationsbearbeitung unter Berücksichtigung von Konfidenz-Zahlen.* Diplomarbeit in Mathematik. Katholische Universität Eichstätt-Ingolstadt, 2009.

[3] Markus Huber, Christian Kölbl, Robert Lorenz, Ronald Römer, and Günther Wirsching. Semantische Dialogmodellierung mit gewichteten Merkmal-Werte-Relationen. In Rüdiger Hoffmann, editor, *Elektronische Sprachsignalverarbeitung 2009. Tagungsband der 20. Konferenz. Dresden, 21. bis 23. September 2009*, volume 53 of *Studientexte zur Sprachkommunikation*, pages 25–32. TUDpress, September 2009.

[4] Markus Huber, Christian Kölbl, and Günther Wirsching. Gewichtete endliche Transduktoren als semantische Träger. In Bernd Kröger and

Peter Birkholz, editors, *Elektronische Sprachsignalverarbeitung 2011. Tagungsband der 22. Konferenz. Aachen, 28. bis 30. September 2011*, volume 61 of *Studientexte zur Sprachkommunikation*, pages 176–183. TUDpress, September 2011.

[5] Mehryar Mohri. Weighted Automata Algorithms. In Manfred Droste, Werner Kuich, and Heiko Vogler, editors, *Handbook of Weighted Automata*, Monographs in Theoretical Computer Science, pages 213–254. 2009.

[6] Günther Wirsching, Christian Kölbl, and Markus Huber. The confidence-probability semiring. Technical report, Angewandte Informatik, Universität Augsburg, 2010.

Missing Pieces of Code

This section gives the pieces of code that were not defined before but are needed to get the fully compilable source code.

Standard Operations for Type `double`, Arctic Semiring

First the standard operations of the arctic semiring are instantiated for the `double`-type.

23a ⟨*Standard type and operations (arctic weight)* 6b⟩+≡ (7b) ◁6b

```
inline ArcticWeightTpl<double> Plus(const ArcticWeightTpl<double> &w1,
                                    const ArcticWeightTpl<double> &w2)
{
  return Plus<double>(w1, w2);
}

inline ArcticWeightTpl<double> Times(const ArcticWeightTpl<double> &w1,
                                     const ArcticWeightTpl<double> &w2)
{
  return Times<double>(w1, w2);
}

inline ArcticWeightTpl<double> Divide(const ArcticWeightTpl<double> &w1,
                                      const ArcticWeightTpl<double> &w2,
                                      DivideType type = DIVIDE_ANY)
{
  return Divide<double>(w1, w2);
}
```

Completion of Class `ProbabilityWeightTpl<T>`

Now we give the remaining parts of the class `ProbabilityWeightTpl<T>`.

23b ⟨*Constructors of class* `ProbabilityWeightTpl<T>` 23b⟩≡ (13a)

```
ProbabilityWeightTpl() :
  BaseType()
{
}

ProbabilityWeightTpl(T f) :
  BaseType(f)
```

```
{
}

ProbabilityWeightTpl(const WeightType &w) :
  BaseType(w)
{
}

explicit ProbabilityWeightTpl(const BaseType &w) :
  BaseType(w)
{
}
```

24a ⟨*Stuff specific to* ***OpenFst*** *(probability weight)* 24a⟩≡ (13a)

```
using BaseType::Value;

typedef WeightType ReverseWeight;

ReverseWeight Reverse() const
{
  return *this;
}

WeightType Quantize(float delta = kDelta) const
{
  return WeightType(BaseType::Quantize(delta));
}
```

24b ⟨*Member functions defining the basic set (probability weight)* 14a⟩+≡ (13a) ◁14a

```
static const WeightType NoWeight()
{
  static const WeightType no(BaseType::NoWeight());

  return no;
}
```

24c ⟨*Member functions giving the identities (probability weight)* 24c⟩≡ (13a)

```
static const WeightType &One()
{
  static const WeightType one(BaseType::One());
```

```
  return one;
}

static const WeightType &Zero()
{
  static const WeightType zero(BaseType::Zero());

  return zero;
}
```

Eventually all the code is put together into one header-file.

25a ⟨*Programme/confidence-probability-semiring/probability-weight.h* 25a⟩≡

```
#ifndef _PROBABILITY_WEIGHT_H_INCLUDED
#define _PROBABILITY_WEIGHT_H_INCLUDED

#include <fst/float-weight.h>

namespace fst
{
  ⟨Layout of class ProbabilityWeightTpl<T> 13a⟩
}

#endif
```

Division, Confidence-Probability Semiring

Now we show the dummy implementation of the division function.

25b ⟨*Division (confidence-probability weight)* 25b⟩≡ (19a)

```
template<class T>
inline ConfidenceProbabilityWeightTpl<T>
  Divide(const ConfidenceProbabilityWeightTpl<T> &w,
         const ConfidenceProbabilityWeightTpl<T> &v,
         DivideType type = DIVIDE_ANY)
{
  LOG(FATAL) << "ConfidenceProbabilityWeight::Divide: "
                "Division is not implemented";

  return ConfidenceProbabilityWeightTpl<T>::One(); // should not happen!
}
```

Standard Operations, Confidence-Probability Semiring

And eventually we show the standard operations of types `float` and `double` for the confidence-probability semiring whereat also the additional template functions are instantiated for these types.

26 ⟨*Standard type and operations (confidence-probability weight)* 19b⟩+≡ (19a) ◁19b

```
inline ConfidenceProbabilityWeightTpl<float>
  Plus(const ConfidenceProbabilityWeightTpl<float> &w,
       const ConfidenceProbabilityWeightTpl<float> &v)
{
  return Plus<float>(w, v);
}

inline float mue(const ConfidenceWeightTpl<float> &c)
{
  return mue<float>(c);
}

inline ProbabilityWeightTpl<float>
  operator*(const ConfidenceWeightTpl<float> &a,
            const ProbabilityWeightTpl<float> &x)
{
  return operator*<float>(a, x);
}

inline ProbabilityWeightTpl<float>
  operator/(const ProbabilityWeightTpl<float> &p,
            const float &c)
{
  return operator/<float>(p, c);
}

inline ConfidenceProbabilityWeightTpl<float>
  Times(const ConfidenceProbabilityWeightTpl<float> &w,
        const ConfidenceProbabilityWeightTpl<float> &v)
{
  return Times<float>(w, v);
}

inline ConfidenceProbabilityWeightTpl<float>
  Divide(const ConfidenceProbabilityWeightTpl<float> &w,
```

```
         const ConfidenceProbabilityWeightTpl<float> &v,
         DivideType type = DIVIDE_ANY)
{
  return Divide<float>(w, v);
}

inline ConfidenceProbabilityWeightTpl<double>
  Plus(const ConfidenceProbabilityWeightTpl<double> &w,
       const ConfidenceProbabilityWeightTpl<double> &v)
{
  return Plus<double>(w, v);
}

inline double mue(const ConfidenceWeightTpl<double> &c)
{
  return mue<double>(c);
}

inline ProbabilityWeightTpl<double>
  operator*(const ConfidenceWeightTpl<double> &a,
            const ProbabilityWeightTpl<double> &x)
{
  return operator*<double>(a, x);
}

inline ProbabilityWeightTpl<double>
  operator/(const ProbabilityWeightTpl<double> &p,
            const double &c)
{
  return operator/<double>(p, c);
}

inline ConfidenceProbabilityWeightTpl<double>
  Times(const ConfidenceProbabilityWeightTpl<double> &w,
        const ConfidenceProbabilityWeightTpl<double> &v)
{
  return Times<double>(w, v);
}

inline ConfidenceProbabilityWeightTpl<double>
  Divide(const ConfidenceProbabilityWeightTpl<double> &w,
```

```
          const ConfidenceProbabilityWeightTpl<double> &v,
          DivideType type = DIVIDE_ANY)
{
  return Divide<double>(w, v);
}
```